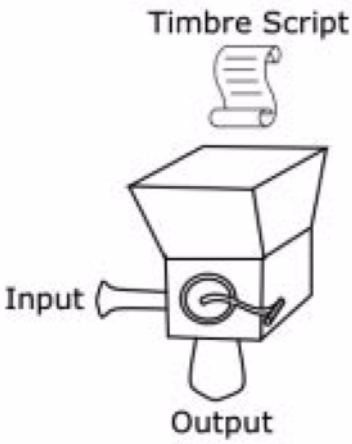
Timbre LC



User Manual for V.14

PREFACE

This manual is intended to provide instruction on how to use Timbre. Timbre runs on Macs and Windows and Unix or Linux platforms. V.11 is a big change in that all words have been changed to lower case.

The information contained in this document is subject to change without notice.

No warranty is made of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. There is no liability for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Copyright TimbreWorks Engineering 1990-2006

Table Of Contents

1 Introduction **5**
Rules 6
Match Phrase 6
Action 7
New Phrase 7
Syntax 7
Rule Sets 8
Syntax 8
Input Flow 8
Conditional Ruling 10
FrameWorks 10
Fundamental Forms 10
Translation 11
Applications 12
An Implementation 12

2 Writing Rules **14**
Simple Rules 15
Translating Input Text 15
Timbre as a Linguistic Solver 15
Parsing 15
Inline States 16
Conditional Substitution 16
Phrases in Forth 16

3 Writing Rule Sets **17**
Rule Sets as States 18
Rule Set Stack 18
Super States 18

4 Writing FrameWorks **19**
Application Stripdown 20
C Parser 20
As a Philosophy 20
. 20

5 Writing Applications **21**

Top Down 23
Bottom Up 23
Middle Out 23
Evolution 23

6 Debugging 24
Rules 25
Rule Sets 25
Frame Works 25
Applications 25

7 FrameWorks 26
C Code Walker 27
Word Walker 27
Noise Parser 27
Stack Verifier 27
68030 Assembler 27

8 Timbre Glossary 28
Stacks 29
Alu 30
Test 30
Memory 32
Dictionary 34
Environment 35
Compilers 39
External Interface 40
File Interface 42
Active Object Loader 44
Library 45
File as an output sink 49
Timbre Engine 50

1
Introduction

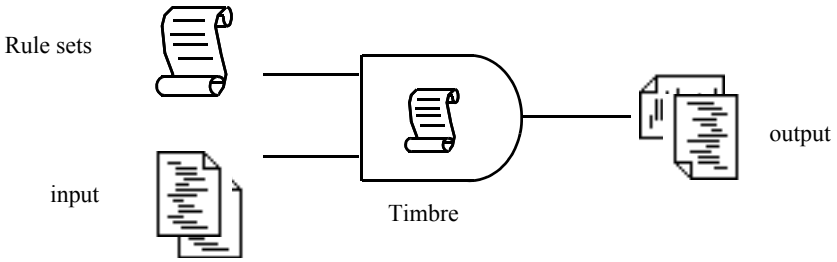
In This Chapter

- *Rules*
- *Rule Sets*
- *Input Flow*
- *FrameWorks*
- *Applications*
- *An Implementation*

Timbre is a general purpose translation engine, whose components are:

1 a pattern matcher for matching text phrases
2 a stack engine for interacting with a computer
3 a rule compiler for defining behaviour

This translation engine may be thought of as a language gate with two inputs and one output. One input controls the behaviour of the gate, and is fed rule sets. The other input is the data. The output is the modified data.



Meta-rules are rules which learn from the input and can edit the rule base. This allows Timbre to adapt to the input.

Rules

A rule is the basic component of behaviour to Timbre. Rules belong to rule-sets. A rule is three groupings of text on one line representing:

1 Match Phrase - when this phrase matches up with the input, the rule executes.
2 Action - action to take when a rule executes.
3 New Phrase - what phrase to stuff back into the input.

The three groups of text are separated by curly, square and curly brackets:

```
{ match phrase }[ action ]{ new phrase }
```

or by vertical bars, square brackets and vertical bars:

```
| match phrase |[ action ]| new phrase |
```

The alternate syntax allows for curly brackets to be included in rules. The two syntactic rules may be intermixed freely.

Match Phrase

A match phrase consists of a group of words and/or symbols which will be looked

for in the input. Once this phrase is found in the input stream, the rule will execute.

The text in the phrase is interpreted as words with blanks between them. The input is parsed with blanks between words except when a rule choses to parse differently.

If the match phrase of a rule is a subset of another rule then the rule with the longer match phrase will execute. If there are two rules with the same match phrase, then the first one found will execute.

Action

An action consists of a group of stack engine instructions[1]. New phrases may also be added as part of an action.

New Phrase

A new phrase is the text that will be stuffed into the input. This new input will then be used for matching rules. This allows a rule to be factored into common components which are self documenting.

Syntax

Rules may be built using curly brackets or vertical bars to delimit the phrases. Either the action or the new phrase may be left out. New phrases can appear inside of an action. Multiple match phrases may be included within a rule.

```
{ match phrase}[ actions { new phrase } ]{ new phrase
```

one or more — either: action, new phrase, or both

alternate syntax:

```
| match phrase |[ actions | new phrase | ]| new phrase
```

Syntax Examples

The following examples illustrate some of the syntactical forms:

```
{ match phrase }[ action ]
{ match phrase }{ new phrase }
```

1 See “botKernel Programmer’s Manual” for more details on stack engine instructions. See the Glossary of this manual for the stack engine API.

```
{ match phrase 1 }{ match phrase 2 }{ match phrase 3 }[ action ]
{ match phrase }[ action { new phrase } ]
{ match phrase }[ action ]{ new phrase }
{ match phrase }[ action { new phrase 1 }  ]{ new phrase 2 }
```

Rule Sets

A rule set is a collection of rules functionally grouped together. Rule sets are declared first and then rules are added to them. Rule sets to be matched to the input are pushed onto the rule stack. Only the rule sets on the rule stack are used to match against the input. By factoring rules into different rule sets, you can create context and run state machines on the input. Super states can be achieved by keeping common rule sets on the rule stack. The rule stack is modified by the actions of rules.

Syntax

Rule sets are declared by the word RULE-SET. It creates the necessary data structures to support a rule set.

```
RULE-SET interpreting
```

A rule set is made available by putting it on the rule stack:

```
ruleset1 RULES
```

Once it is on the rule stack, rules may be defined.

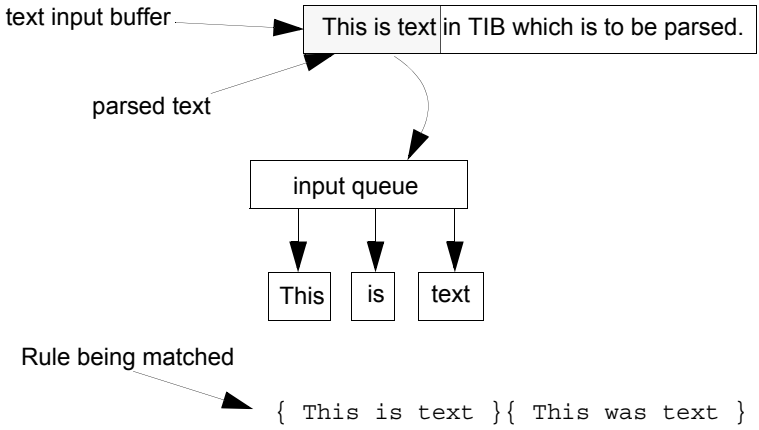
Several rule sets may be placed on the rule stack or taken off by:

```
ruleset1 >RULES  ruleset2 >RULES
RULES> DROP
```

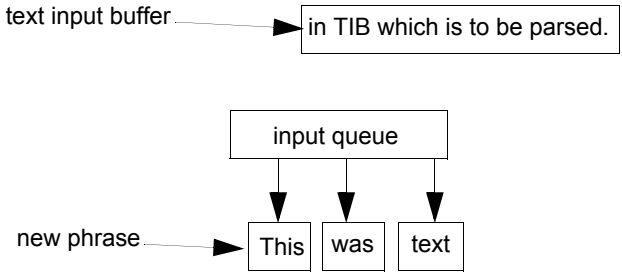
Input Flow

Text flows from the text input buffer (TIB) into an input queue. This input queue is filled with words parsed from TIB. Rules are matched against the words in the input queue. If a rule needs more words to complete its match phrase, then they are parsed from the TIB. When a rule's match phrase matches, all the matching words are dropped from the input queue and the rule is executed. Any remaining words and new phrases

in the input queue are used to match rules again.



After this rule has fired, we have the following picture:

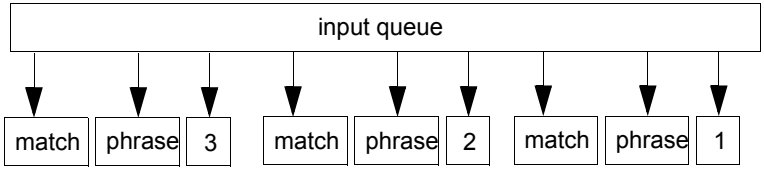


In a rule which has more than one new phrase to stuff, the last stuffed phrase is the first accessible phrase in the input:

When this rule fires,

```
{ match phrase }[ { new phrase 1 } { new phrase 2 } ]{ new phrase 3 }
```

The input queue looks like:



Conditional Ruling

All looping, conditional or iterative, recursion and co-routine program constructs are supported by conditionally substituting a new phrase inside an action:

Example: iterative looping

```
{ until 0 }[ ?DUP  IF  1 -  { until 0 }  ENDIF ]
```

Example: conditional looping

```
{ until false }[ IF  { until false }  ENDIF ]
```

Example: recursion

```
{ this rule }[ MORE?  IF  { this rule }  ENDIF ]
```

Example: co-routines

```
{ that rule }[ OTHER?  IF  { other rule }  ENDIF ]
{ other rule }[ THAT?  IF  { that rule  }  ENDIF ]
```

FrameWorks

FrameWorks are collections of rule sets which may be used to build Applications. They are developed from an application, or during its building. It is a frame of rule sets that works (i.e. noBuGs)! Examples include:

- noise parser
- partial 68030 assembler
- infix math parser
- RPN math parser

Fundamental Forms

Here are the fundamental forms of all Timbre rules for all subsequent Frameworks and scripts augmented with a few illuminating comments:

(1) { } recognition
(2) [ ] control
(3) { }{ } substitution
(4) { }[ ] action
(5) { }[ ]{ } translation
(6) { }[{ }] controlled substitution
(7) [ ]{ } pattern generator
(8) [{ }] controlled pattern generator

(9) [{ }]{ } semi-controlled pattern generator
(10) { }[{ }]{ } transmorf

Recognition

- pattern is parsed out of the input
- it is in the dictionary
- an empty pattern matches everything else, it is the default rule

Control

- it runs the machinery when there is no input
- self sustaining
- default action is to continue to parse words from open files using blanks as delimiters

Substitution

- just altering the flow
- optimizations
- self sustaining

Action

- from the event, action, change state model
- states are supported as well as context nesting

Translation

- recognition, control and substitution, what more do you need?
- self sustaining

Controlled Substitution

- oh yeah, gotta control some of those substitutions
- self sustaining

Pattern Generator

- gotta have one, may as well be in concert
- continuous

Controlled Pattern Generator

- this is creativity, only appreciated by others
- self destroying
- second name: creativity

Semi-Controlled Pattern Generator

- nice to have some shorthand in the creation
- self destroying

Transmorf

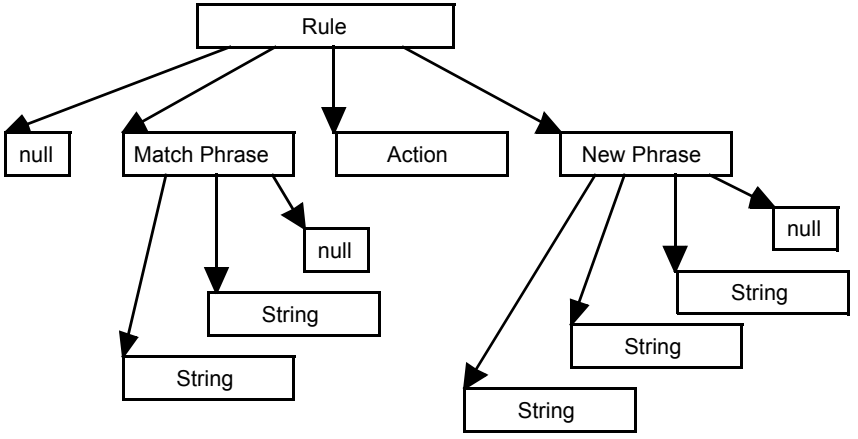
- I see, I channel, I add
- I can do it all, every thing else is just shorthand

Applications

Applications may be built from FrameWorks or from new rule sets. When building the rule sets consider what states are required in the solution. Simple states may be done with a single rule while more complex states would require a complete rule set change. Some applications are required to learn from their input. To learn, certain rules called meta-rules create new rules and or rule sets. A meta-rule can also create new meta-rules.

An Implementation

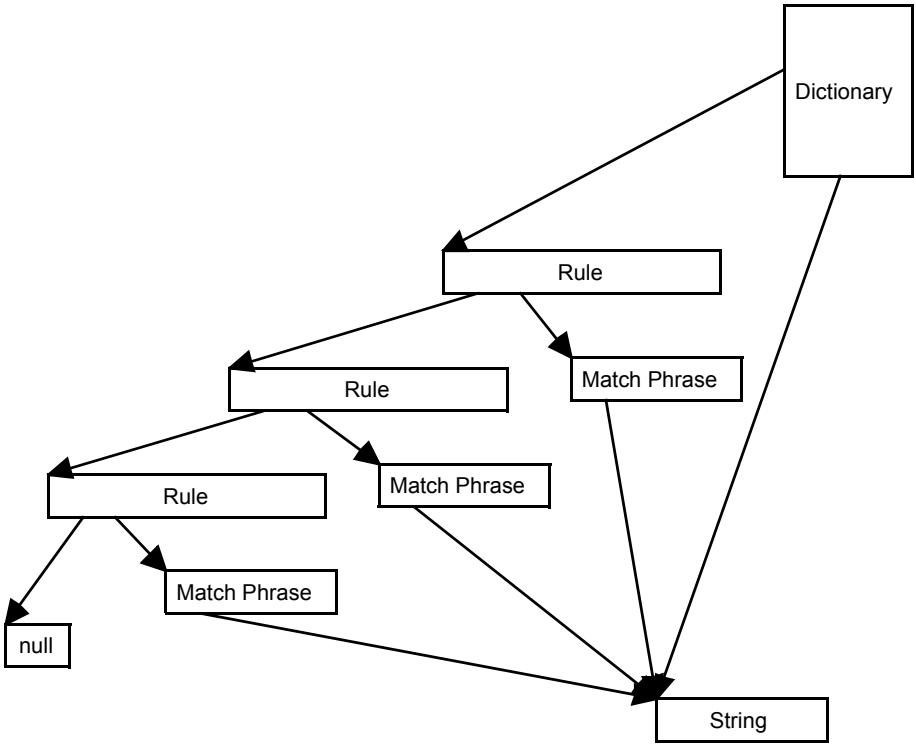
A closer look at the relationships between rules and their components brings us to the nuts & bolts level. An implementation in software of the Timbre engine lets us examine what components can be used to build rules.



A rule is an array of 3 pointers. A match phrase or a new phrase is an array of pointers to strings terminated with a null. An action is code which can be exe-

cuted by the local stack engine. A string is an array of characters with the count in front. The strings are kept unique by keeping them in a hashed dictionary which provides fast lookup. By maintaining a single copy of an input string, it rapidly procedes through the Timbre engine with address comparisons clicking most of the time with an occasional string compare. New input or noisy input will cause more string compares.

For fast rule recognition, the rules are indexed into a dictionary on the first string of the Match Phrase. Rules with a common first string are link listed together.



2

Writing Rules

In This Chapter

- *Simple Rules*
- *Inline States*
- *Conditional Substitution*
- *Phrases in Forth*

Simple Rules

Timbre rules involve an iterative thinking process. As you write out rules, you describe, understand and solve the problem you are working through. It is best to minimize your actions and stay in the editor continuously refactoring until you feel you have a firm grasp of the problem. Since Timbre is text oriented, it is a lot like writing a small story.

Writing rules breaks down into two categories. In the first type, text input is being translated. In the second type, input, which isn't necessarily text, is translated linguistically.

Translating Input Text

Translating input text usually has a structure which consists of a lot of parallel phrases. If you are encountering varied text phrases in the input, write a phrase that you expect to translate then write its substitution. This might involve using different forms of Timbre as described in "Fundamental Forms" on page 10.

Timbre as a Linguistic Solver

Using Timbre as a linguistic solver involves building a hierarchy of rules. The simplest way to start writing rules is to start at the highest level and describe your problem linguistically without any actions. If you are using Timbre in a mode where there is no text in the input but you are using Timbre linguistics to solve a problem, then write the phrase you expect to solve and then write the solution as a substitute phrase. You move down in the hierarchy by making sub-rules that solve portions of the initial substitute phrase by creating rules with match phrases and substitute phrases. At some point the phrases are atomic and are then described as actions linking them to the underlying computing machinery.

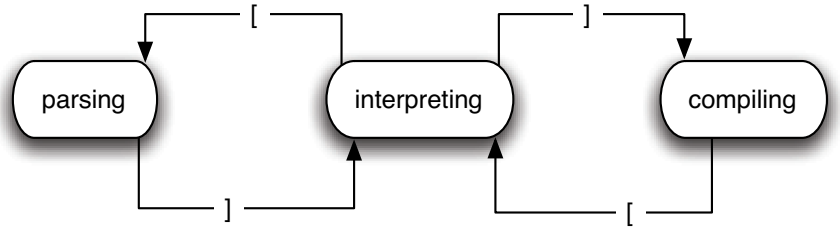
Parsing

To change from an interpreting state to a compile state the right bracket "]" is used and "[" is used to return to the interpret state from the compile state. In the compile state special words have been given the power to temporarily be in the interpret state while the compile state is invoked.

Timbre adds one more state: Parsing. The same two symbols are used change states. The parsing state is transitioned to by using the left bracket "[" and then back to the interpret state with right bracket "]".

The default parser in Timbre is to use ASCII blanks to delimit non-blank sequences of characters from the input stream as strings. Different parsers are built for any rule set

by using the [ ] sequence.



The default parser can be described as:

[bl word here unique inputq push]

Inline States

```
{ count down }[ ?dup if 1 - { count down } then ]
```

This rule creates a temporary inline state that counts down the top element on the data stack till zero, and then stops.

Conditional Substitution

Phrases in Forth

```
: hello-world  | hello world | ;
```

In the above Forth definition the phrase “hello world” is inserted into the input stream as two words.

3

Writing Rule Sets

In This Chapter

- *Rule Sets as States*
- *Rule Set Stack*
- *Super States*

Rule Sets as States

Rule Set Stack

Super States

4

Writing FrameWorks

In This Chapter

- *Application Stripdown*
- *As a Philosophy*

FrameWorks are the scaffolding upon which applications are built from. They are a collection of rule sets with a basic behaviour. A technique for parsing, or an algorithm for code certifying. They are either built by distilling from applications or with original intent.

Application Stripdown

Strip extra functionality not related to a primary state diagram of the system that is not specific to a set of rules, is the distilled essence, or a FrameWork(s) for an application. A framework creates a state diagram of linguistic terms

C Parser

This framework was distilled from an application which was used to scan C code using rules to transit linguistic states. Addition of rules can create a host of applications involving C structure like code. Simple compilers or support for other linguistic characterstics can be added.

Parser State Diagram

As a Philosophy

Good frameworks are the cornerstone of design amplification. Each framework captures a linguistic diagram which represents the set of states and transitions necessary to walk the states. Some states might be isolated and have zero transitions or more, in or out. It is good practice to generate frameworks from an application to give back to Timbre Zone.

Different classes of Frameworks in clude a capture architecture, subsumption, motor schema, pattern finders, text parsers, etc.

5

Writing Applications

In This Chapter

- *Top Down*
- *Bottom Up*
- *Middle Out*
- *Evolution*

Top Down

Bottom Up

Middle Out

Evolution

6

Debugging

In This Chapter

- *Rules*
- *Rule Sets*
- *Frame Works*
- *Applications*

Rules

Rule Sets

Frame Works

Applications

7

FrameWorks

In This Chapter

- *Noise Parser*
- *Stack Verifier*
- *68030 Assembler*

C Code Walker

Word Walker

Noise Parser

Stack Verifier

68030 Assembler

8

Timbre Glossary

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

This glossary contains a stack comment and a simple explanation of each word available in Timbre. The words are grouped according to usage. For an alphabetical listing, refer to the index.

Cell

cell	(-- n) number of bytes in a memory cell
cells	(n -- m) number of bytes in n cells

Stacks

Data Stack

swap	(m \ n -- n \ m) swap the top two items of the stack
dup	(m -- m \ m) duplicate the top item of the stack
drop	(m --) drop the top item of the stack
nup	(m \ n -- m \ m \ n) duplicate the second stack item (sounds like dup)
nip	(m \ n -- n) drop the second item on the stack
over	(m \ n -- m \ n \ m) copy the second item to the top of the stack
tuck	(m \ n -- n \ m \ n) tuck a copy of the top item under the second item
2dup	(m \ n -- m \ n \ m \ n) duplicate the top two items
2drop	(m \ n --) drop the top two items
?dup	(n -- [n] \ n) duplicate the top item of the stack if it is not 0
sp!	(? --) empty the data stack
depth	(-- n) return the current depth of the data stack
sp0	(-- a) pointer to bottom of the data stack

Return Stack

r>	(-- m) pop the top item off the return stack
>r	(m --) push n onto the return stack
r	(-- m) get a copy of the top item on the return stack
rp!	(--) empty the return stack
rdepth	(-- n) return the current depth of the return stack
rp0	(-- a) pointer to bottom of the return stack

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Alu

Logic

and	(m \ n -- p) AND n with m
or	(m \ n -- p) OR n with m
xor	(m \ n -- p) XOR n with m
not	(m -- n) invert all the bits of n

Shift

2*	(m -- n) shift m left one bit and set bit 0 to 0
2/	(m -- n) shift m right one bit but don't change msbit
u2/	(m -- n) shift m right one bit and set msbit to 0
shift	(n \ m -- n) shift n m bits left or right; -m is left

Math

+	(m \ n -- p) add n to m
-	(m \ n -- p) subtract n from m
negate	(m -- n) take the two's complement of m
/mod	(n \ m -- r \ q) unsigned divide with remainder. devide n by m.
/	(n \ m -- quot) unsigned divide
mod	(n \ m -- rem) modulus
*	(n \ m -- nm*) signed multiply

Test

Flags

no	(-- f) the false flag, 0.
yes	(-- t) the true flag, -1.

Zero Test

0=	(n -- flag) return yes if n is 0, else return no
0<	(n -- flag) return yes if n is negative, else return no
non?	(f -- f') French equivelant for 0=

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Compare

=	(n \ m -- flag) return yes if n is equal to m, else return no
<	(n \ m -- flag) return yes if n is less than m, else return no
>	(n \ m -- flag) return yes if n is greater than m, else return no
u<	(n \ m -- flag) same as < but unsigned and used typically for addresses
u>	(n \ m -- flag) same as > but unsigned and used typically for addresses
differ?	(a \ a -- a+ \ a+ \ f) compare bytes and increment to next bytes
bits-differ?	(a \ a \ mask -- a+ \ a+ \ f) differ? using a bit mask
same?	(a1 \ a2 \ length -- flag) compare length bytes at a1 to a2
compare	(string1 \ string2 -- flag) compare two count-prefixed strings
memcmp	(a \ b \ length -- flag:[-c 0 +c]) compare two strings and return difference of first different character

Absolute, MinMax

abs	(n -- n) take the absolute value of n (note: 8000 abs is 8000)
max	(n \ m -- p) return the signed maximum of n or m
min	(n \ m -- p) return the signed minimum of n or m
umax	(a \ b -- c) return the unsigned maximum of a and b

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Memory

Cells

@	(a -- n) read the cell at address a
!	(n \ a --) store n into the cell at address a
@+	(a -- n\ a+) read the cell at address a and increment a to the next cell
!+	(n \ a -- a+) store a word at address a and increment a to the next cell
@-	(a -- n\ a-) read the cell at address a and decrement a to the previous cell
!-	(n \ a -- a-) store a word at address a and decrement a to the previous cell
+!	(n \ addr --) increment the value in the cell at address a by n

Bytes

c@	(a -- c) fetch a byte from address a
c!	(c \ a --) store a byte at address a
c@+	(a -- c\ a+) fetch a byte from address a and increment a by 1
c!+	(c \ a -- a+) store a byte at address a and increment a by 1
c@-	(a -- c\ a-) fetch a byte from address a and decrement a by 1
c!-	(c \ a -- a-) store a byte at address a and decrement a by 1
c+!	(n \ addr --) increment the value in the byte at address a by n

Bits

+bits	(bits \ addr --) set bits of the byte at address a
-bits	(bits \ addr --) clear bits of the byte at address a

Short

w@	(addr -- n) fetch two bytes at address
w!	(n \ addr --) store two bytes to address

Queues

queue	(#cells --) create a queue with #words entries
0q	(q --) remove all items from a queue
q	(q -- n) get a copy of first value in queue
p	(q -- n) get a copy of the last value in the queue
q?	(q -- n) return number of items in a queue
q>	(q -- n) return first value in queue
>q	(n \ q --) append n to end of queue
rotateq	(q \ n --) rotate n elements through queue q

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Queues

transferq	(srcq \ dstq \ n --) transfer n elements form source queue to destination queue
push	(n \ q --) push n into queue; same as >q
pop	(q -- n) pop an item out of the bottom of a queue; used with PUSH to implement a stack
stuff	(n \ q --) stuff n back into a queue; used with pull to implement a stack
pull	(q -- n) pull n out of a queue; used with push to implement a queue; same as q>

Memory Tools

bl	(-- char) the ascii value of blank, hex 20
cmove	(src \ dest \ n --) move n bytes from src to dest
<cmove	(src \ dest \ n --) same as cmove but going from high to low memory
move	(src \ dest \ n --) move n cells from src to dest
fill	(addr \ n \ char --) fill n bytes of memory with char starting at address addr
erase	(addr \ n --) fill n bytes of memory with 0 starting at address addr
blanks	(addr \ n --) fill n bytes of memory with 20H starting at address addr

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Dictionary

Memory Management

dp	(-- addr) variable containing the address of the start of free memory
limit	(-- a) variable containing the address of the end of free memory
here	(-- addr) return the contents of free
allot	(n --) allocate n bytes of memory
,	(n --) store n into free memory and increment free to the next cell
c,	(c --) store c into free memory and increment free by 1
w,	(n --) store n into free memory and increment free by 2
aligned	(--) round free up to an aligned address
align	(a -- a') round a up to an aligned address according to platform

Headers

latest	(-- addr) points to the link field of the most recently defined word
t>link	(tick -- link) convert a tick address to a link address
l>tick	(link -- tick) convert a link address to a tick address
t>name	(tick -- name) convert a tick address to a name address
n>tick	(name -- tick) convert a name address to a tick address
n>link	(name -- link) convert a name address to a link address
l>name	(link -- name) convert a link address to a name address
l>link	(link -- link) traverse from the link of one word to the previous
>body	(tick -- body) convert a tick address to its body address

Dictionary Searching

name?	(string \ name -- flag) compare string and name
search-dictionary	(string \ >list -- link \ yes | -- string \ no) search list for string

Vocabularies

vocabulary	("name" -- v) CREATE a header and add dictionary to it
search-context	(s -- a \ f) search context vocabuluary for string
search-vocab	(s \ v -- s \ f) search a single vocabulary
set-context	(v --) set context to v

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Vocabularies

only	(--) set context back to default of forth and root
definitions	(--) make first searched vocabulary the one to add new words to
listq	(-- q) stack of vocabularies in context for searchings
also	(--) make a copy of the top vocabulary to the vocabulary stack
forth	(--) set search order to built-in vocabularies

Environment

Baron

peasantq	(-- q) address of peasantq
0baron	(--) empty the peasant queue
>baron	(peasant --) append a peasant to the peasantq
baron>	(-- peasant) remove peasant from peasantq
baron	(--) execute a peasant from the peasant queue
kill	(peasant --) remove all occurances of a peasant from the peasantq
run	(peasant --) append a peasant and remove any other occurances in the peasantq
?baron-full	(--) test to see if task queue is full

Output Stream

emitq	(-- q) output queue; used by emit
out	(-- q) variable holding the number of characters output since cr
flush-emits	(--) run baron until emitq is empty
?wait	(--) if the emitq is full, run the baron until the emitq is not full
emit	(char --) stuff a character in the emitq
emit?	(c -- c|0) output a single character and return zero if successful

Output Formatter

cr	(--) emit a linefeed and a cursor return
?cr	(--) call cr if out is not 0
space	(--) emit a space
spaces	(n --) emit n spaces
count	(addr -- addr' \ count) read a byte and increment addr by 1
type	(addr \ count --) emit n characters from address addr

Number Output

base	(-- addr) variable containing current base for number I/O
set-base	(a \ c -- a' \ c') check for prefixes: 0X, 0x, 0C, 0c, 0B or 0b
bin	(--) set number base to 2, binary
oct	(--) set number base to 8, octal
hex	(--) set current base to hex
decimal	(--) set current base to decimal
pad	(-- addr) a scratch pad area for number conversion
hold	(char --) prepend char to current number string
<#	(--) start number to string conversion
#	(n -- n) convert the ls digit of n to a character and prepend it to the current number string
#s	(n -- n) convert all digits until n is 0
sign	(m \ n -- n) prepend sign of m to current number string
#>	(n -- addr \ count) end number conversion and return address and length
.r	(n \ m --) output n to screen, right justified in field m
.	(n --) output n to screen
.b	(n --) print out n in binary
.d	(n --) print out n in decimal
.h	(n --) print out n in hexadecimal

Input Stream

keyq	(-- q) keyboard input queue; checked by key
key?	(-- flag) number of items in keyq
key	(-- char) get a character from the keyq
0keys	(--) empty the key queue
key-ask	(-- c|0) query for nonblocking input

Text Input Buffer

in	(-- a) index into text input buffer for parsing
tib	(-- a) points to text input buffer
0tib	(--) reset tib pointer and index
input	(-- addr) current address within input buffer
+in	(addr --) set input pointer to addr

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Input Scanner

skip	(char --) skip over any bytes in tib that equal char
scan	(char --) scan tib until char is found
scan-for	(char -- f) scan tib for char and return flag
parse	(char --) copy all characters till char is encountered, to here

Error Recovery

abort	(--) reset the return stack
error	(--) print out any string at here and call abort
?error	(flag --) call error if flag is non-zero

Number Conversion

lowercase	(c -- c') make an alphabetical character lower case
negative?	(a -- f) return true if first character is a minus sign
prefixed?	(a \ c -- f) return true if string longer than two start with zero
skip-prefix	(a \ c \ n -- a' \ c') skip n characters
digit	(char -- n \ flag) try to convert a character to n and return a flag
digits?	(a \ n -- f) check digits to see if they are digits in the current base
digits	(a \ n -- a+ \ m) convert n digits and abort on error
number?	(string -- f) check string to see if it is a number in the current base
number	(string -- n) convert a string to a number n or die trying

Compiler

compiling	(-- addr) variable containing current state of compiler
]	(--) set state to compile
[	(--) set state to interpret
late	(tick -- tick') return the latest definition of the word
compile	(token --) compile the token into the dictionary
?execute	(tick \ status --) execute or compile tick address
literal	(n --) compile a number into the dictionary

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Inner Interpreters for Cells

>i	(n --) push n into index register for use with -branch
i>	(-- n) pull n from the index register
lit	(-- n) push contents from following cell onto data stack
branch	(--) branch to address in following cell
0branch	(n --) branch if stack is 0
-branch	(--) decrement index register and branch if not -1
case-branch	(n \ m -- [n]) branch constructuct for case

Outer Interpreter for Text

(	(--) skip all characters up to and including)
\	(--) skip all characters till end of line
word	(char --) copy all bytes up until char to here skipping leading chars
char	(-- n) return the next non-blank character in the input
input?	(-- f) returns yes if there is more input and will pull more lines from file if available
?char	(-- c) return a character from multi-line input
?word	(c --) parse a “c” delimited word from multi-line input
find?	(-- cfa \ status | -- string \ no) try to find next word in input stream
?find	(-- cfa) try to find next word in input stream or die trying
interpret	(--) interpret the input stream pointed at by tib and in till 0
`	(-- n) return the ASCII value of the next letter
execute	(a --) execute code at address a
execute-word	(? \ s -- ?) look up and execute word in string s
interpret-word	(s --) look up and compile/execute word in string
container	(-- a) array for text evaluation
:noname	(-- tick) start a headless definition returning its execution address
evaluate	(s --) evaluate a text string

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Key Collector

prompter	(-- addr) address of the prompt string
prompt	(--) set the prompt string as the rest of tib
.prompt	(--) output the prompt
collect	(--) collect and echo input characters
collector	(--) collect keyboard input and interpret when cr is input

Control Loop

init	(--) first in initialization chain called by reset
quit	(--) main control loop which continuosly executes baron

Compilers

Control Flow

ahead	(--) branch ahead. Terminated with ENDIF. Same as 0 if.
if	(n --) execute following code if n is non zero. Use with else or endif
else	(--) execute following code for zero conditions from if. Use with if and endif
endif	(--) terminator for else or if
then	(--) terminator for else or if
begin	(--) initiate a loop. Ended with either until, again or while...repeat.
while	(n --) if n is zero, continue execution after repeat otherwise execute following code
repeat	(--) branch back to begin
until	(n --) branch back to begin if n is zero
again	(--) always branch back to begin
for	(n --) execute code enclosed by next, n times. 0 really means zero times.
next	(--) decrement index until 0 and branch back to for
exit	(--) return to caller of current word
'	(-- cfa) return the code field address of the next word in the input stream

Strings

string	(c -- string) return a string terminated by c
"	(-- s) compile a string delimited by a "
."	(--) print out message delimited by "

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Inner Interpreters

vii	(-- ii)	inner interpreter for variables
cii	(-- ii)	inner interpreter for constants
:ii	(-- ii)	inner interpreter for colon definitions
dii	(-- ii)	inner interpreter for does children

Defining Words

header	(--)	create a header for the next word in the input stream
smudge	(--)	hide the latest word from the interpreter
recursive	(--)	reveal the latest word to the interpreter
immediate	(--)	make the latest word executable at compile time
create	(--)	create a word and have it return the address of space after it
<builds	(--)	create a word which has an action defined by does>
does>	(--)	define an action for a word built with <builds
:	(--)	start a word definition
;	(--)	terminate a word definition
constant	(n --)	create a constant and set it to n
variable	(n --)	create a variable and initialize it to n
forget	(--)	remove all words defined after and including the next word in the input stream

External Interface

End of Kernel

version	(--)	print out version of botForth
reset	(--)	reset the kernel
reboot	(--)	initial entry point to botForth
install	(--)	do nothing word for compatibility; to be removed
bye	(--)	exit the kernel

Standard IO

stdin	(--)	peasant for polling kernel input queue
stdout	(--)	peasant for polling kernel output queue

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Standard IO

init-io	(--) initialize input/output
setup-io	(--) setup the io interface for non blocking
restore-io	(--) restore io to entry conditions
sys	(“commands” --) the rest of the line is passed as a command line to the underlying operating system
sleeps	(-- a) system variable to control amount of time spent sleeping each cycle. Set to 1 for maximum speed but increase if IO is a problem.

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

File Interface

File Access

byte-pointer	(-- a) variable containing byte index into a file for reading/ writing
line-no	(-- a) variable containing current line number in text file
file	(-- a) variable which contains the file handle
file-status	(-- a) vartiable which contains the status of the open file
file-size	(-- a) variable containing size of file just opened
new	(--) use before file to create a new file
modify	(--) use before file to allow writing to a file
binary	(--) use before file to allow binary access to a file
open	(s --) open a file named string with whatever above modifiers
?file	(--) abort if file-status is set
close	(--) close the open file
remove	(name --) delete file of name
rename	(old name \ new name --) rename a file
locate-file	(s -- f) return yes if the file exists and can be opened
file?	(-- n) return 0 if there are no files open
close-all	(--) close all open files
push-file	(--) push current file parameters onto file stack
pop-file	(--) pop the file parameters off the file stack and make it the current file

File Contents

read	(-- c) read a byte from a file and increment the byte pointer
write	(c --) write a byte to the file and increment the byte pointer
read-line	(a \ n -- n) try to read n bytes from file to address a and return number read
write-line	(a \ n --) write n bytes from address a to file and append an end of line
read-cell	(-- n) read a cell from a file
write-cell	(n --) write a cell to a file
read-bytes	(a \ n --) read n bytes from a file to address a
write-bytes	(a \ n --) write n bytes to a file from address a

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Block Access

bytes	(-- n) return number of bytes in a block
block	(n -- a) return address of block n
update	(--) mark current accessed block as changed so it will be written back
flush-blocks	(--) flush any modified blocks

Directories

fullfilename	(-- a) address containing path and last file name
pathlength	(-- a) variable containing length of current working directory
rootlength	(-- a) variable containing length of root directory
pwd	(--) print out current directory
set-root	(s --) set current directory to string
add-path	(s -- s') concatenate string to the end of current path
cd	("path" --) set the root directory Useage: cd botForth in C/Forth to C/
wd	("path" --) set working directory with reference to root Useage: wd A/

Source Loader

input-echo	(-- a) flag variable. If set to yes, input will be echoed during loading.
input-line	(-- f) copy a line from a file into tib. 0 returned, if not possible.
ld	("name" --) compile a source file. Use: ld file.f
load	("file" --) interpret a file with echo off

Automatic File Loading

auto-script	(--) load INIT.F in current directory
init	(--) initialize autoscripting

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Active Object Loader

Statistics

unknowns	(-- a) number of unknown symbols encountered
0errors	(--) reset error counts
+unknown	(--) increment the number of unknowns
.done	(f --) print out final stats
unknown	(--) error word for unknown symbols

Active Objects

repeat-object	(--) for repeating active-objects
ao-message	(--) print out embedded message
assume#	(s -- n) convert string to a number
interpret-object	(s --) interpret next object in file
activator	(--) active object to interpret file
load-aof	(--) embedded active file linker
ao-branch	(--) branch absolute within the file
ao-0branch	(f --) branch if 0 within the file

File Access

aobj_file	(-- n) FDTYPE for an active object file
read-name	(-- s) read a count prefixed string from file
next-symbol	(-- tick | 0) read next symbol from file
lookup-symbol	(-- tick) next-symbol with error checking
activate-object	(--) activate next object in file
activate-file	(s --) primitive to execute active-object file
execao	(s --) execute active-object file
act	(“object” --) prefix syntax for activating file

Image Relocater

image	(-- a) pointer to start of working image
absolute	(ra -- aa) change relative address to absolute
relative	(aa -- ra) change absolute address to relative
immediate?	(tick -- f) check immediate bit in ticked word
resolve-symbol	(-- a) resolve symbol to an address
load-image	(a --) read image from file to memory

Image Relocater

external-value	(--) resolve an external value
external-ref	(--) resolve an external reference
inlink	(--) resolve a reference into image
outlink	(--) resolve a reference out of image
outlist	(--) resolve a link list to an external reference
internal-ref	(--) resolve an image internal reference
internal-value	(--) resolve an image value
table-entry	(a --) resolve a table entry

Library

Debugging Tools

.line	(a -- a+) print out a line for dump
.fields	(a --) print out the fields for dump
dump	(a \ n --) dump n lines of 16 bytes from address a
.s	(--) print out the data stack
in-what	(addr -- link | 0) return link of word containing addr
?ascii	(char -- char) convert to . if not printable ASCII
?stop	(--) stop if a key has been pressed
.name	(lf --) givin a link, print the word name
.word	(a --) print name associated with address a
show-peasants	(--) display current peasants in queue
peasants	(--) display all peasants in queue
words	(--) display all words in dictionary
watch-stack	(--) monitors peasants and signals when one of them changes the data stack Use: ‘ watch-stack >baron

Case Construct

case	(-- 0) start a case construct
{	(0. -- 0..) begin a choice
}	(0.. -- 0..) end a choice
endcase	(0... --) end a case construct

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Hashed Dictionaries

dict	(-- a) pointer to current dictionary
string-table	(-- a) address of string table in current dictionary
capacity#	(-- n) capacity of current dictionary
#free	(-- n) number of free spaces left in current dictionary
adjunct-table	(-- a) address of adjunct table in current dictionary
dictionary	(-- dict) create an empty dictionary
free-dictionary	(dict --) release a dictionary
#hash	(-- n) dictionary size
empty	(dict --) empty a dictionary
initialize	(--) initialize new dictionaries
+entry	(--) decrement number of free entries
-entry	(--) increment number of free entries
?adjunct	(--) setup an adjunct table if one doesn't exist
0string	(-- a) zero-length-null-string for replacing deleted entries
used?	(loc -- f) check to see if a location is used
different?	(string \ loc -- f) compare a string to a location
hash	(string -- loc) hash a string to a dicionary location
asckey	(string \ loc -- string \ loc') rehashing algorithm
rehash	(string \ loc -- string \ loc') rehash string for next location
junk	(-- a) junk entry for when there is no dictionary
bump	(string \ loc -- string' \ loc) exchange string with entry of same content
locate	(string \ dict -- loc) lookup up string in dictionary
insert	(string \ dict --) insert string into a dictionary
append	(string \ dict --) append a string into a dictionary
delete	(string \ dict --) delete string from dictionary
find	(string \ dict -- entry | 0) find string in dictionary
adjunct	(string \ dict -- entry | 0) find cell in adjunct table

Dynamic Memory

free	(a --) free an allocated chunk of memory
allocate	(n -- a) allocate a chunk of memory
.ram	(--) print out memory usage

Link Lists

previous	(entry \ head -- prev) find previous entry in list
link	(a \ b --) link two entries
unlink	(b -- a) unlink entry a from b
chain	(head --) link here into the list

Strings

numstr	(-- a) space for writing a number string
$!	(s \ a --) store string s at address a
+$	(s \ a \ n --) append n bytes at a to string s
#>$	(n -- s) convert a number to a tempoary string
$within	(s1 \ a \ n -- a' | 0) look for s1 within n bytes of a

Nested Enviroment Shells

shell-end	(--) reset the return stack but allow data
shell-out	(? --) reset both stacks
shell-abort	(? --) reset both stacks and set error flag
shell	(--) Shell execution for indirect threaded Forth code
shell{	(--) Shell execution for indirect threaded Forth code
}shell	(-- flag) terminate shell
sh!	(--) reset shell stacks

Timeout Facility

timeout	("name" --) create a timeout variable
time	(-- seconds) return the time from start in seconds
timeout?	(a -- f) check a timeout variable to see if it is done
set-timeout	(n \ a --) set a timeout variable for n seconds from now
start	(--) set the start time for end
end	(--) return the length of time from start
pause	(ms --) stop application for ms milliseconds

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Alerts

#arg	(-- n) used to prefix numbers
error-string	(n -- s) return the error string for error number n
w.notice	(s --) print out s as a notification
w.error	(s --) print out s as a error message
w.messages	(..\ n --) print out messages
w.notices	(..\ n --) print out strings and numbers as a notification
w.errors	(..\ n --) print out strings and numbers as an error message

Quick Sort

array	(-- a) points to array of items to sort
>item	(n -- a) index to n^{th} item address in array
string-less?	(i \ pivot -- f) check if item is less than pivot item
unsigned-less	(i \ pivot -- f) check if item is less than unsigned pivot item
less-vector	(-- a) points to compare routine
less?	(i \ pivot -- f) call compare routine
cell-exchange	(loc1 \ loc2 --) exchange two locations
exchange-vector	(-- a) points to exchange routine
exchange	(loc1 \ loc2 --) call exchange routine
partition	(high \ low -- pivot#) create low and high list and a pivot#
rqsort	(high \ low --) recursive part of quicksort
quicksort	(list \ n --) use quicksort to sort list of n items
sort-byadjunct	(dict -- dual array) sort strings according to adjunct values
sort-dict	(dict -- array \ n) sort dictionary into an array of n items
sort	(dict --) sort a dictionary and print out items

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

File as an output sink

Output File Type

platform	(-- a) contains file type to create for output
#dos	(-- n) constant to indicate DOS file type with CR+LF as line ending
#mac	(-- n) indicates Mac file type with CR line ending
#unix	(-- n) indicates Unix file type with LF line ending

Output File Control

output	(-- a) variable for current file id and byte-pointer
filename	(-- a) array containing name of input file
destination	(-- a) array containing name of destination file
trim-blank	(--) remove trailing blanks from filenames
source	(--) switch to source file
sink	(--) switch to sink filel
swap-file	(--) swap top files on the open file stack
dos	(--) use dos file format; CR/LF
dos?	(-- f) return true if platform is dos file type
mac	(--) use Mac file format; CR
mac?	(-- f) return true if platform is mac file type
unix	(--) use Unix file format; LF
unix?	(-- f) return true if platform is unix file type

Output Redirection

file-emit	(--) machine to output emits to a file
>console	(--) switch output back to console
>file	(“file” --) direct output to file
>folder	(s --) select destination file space
file>	(s --) send file s to output

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Timbre Engine

Word Dictionary

version	(--) print out version of Timbre
rule-words	(-- a) points to all words which appear in all rules
ruledict	(-- dict) all words which appear in all rules
default	(-- s) string for finding default rules
parser	(-- s) string for parser "[ ]"
in-rules?	(s -- s' \ f) find string in rule word dictionary
make-word	(s -- name) add new string to dict if not already there
get-word	(-- name) parse out the next word and add it to the rule words dictionary if it is new
names	(-- a) points to dictionary of names
unique	(s -- s') return unique address for given string

Rule Sets

ruleq	(-- q) stack of rule sets for input pattern matching
dictq	(-- q) stack of check rule sets
rules-get	(-- rule-set) get top rule set from rule set stack
put-rules	(rule-set --) push rule set onto rule set stack
rule-set	(--) declare a rule set
rule-sets?	(-- n) how many rule-sets are stacked
next-rule-set	(--) go to the next rule set
reset-rules	(--) reset the rule set stack
default-rule	(-- >rule) get default rule from top rule-set
parser-rule	(-- >rule) get parser rule from top rule-set
rules>	(-- rule-set) push a rule set
>rules	(rule-set --) pop a rule set
change-rules	(rule-set --) replace current rule-set
rules	(ruleset --) empty rule set stack and set to ruleset
rssq	(-- q) rule stack states stack to nest rule sets
copy-rules	(--) push a copy of current rule sets to rssq
push-rules	(--) push the current rule sets to the rssq
pop-rules	(--) pop the last rule set state back as current

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Rule Builder

inputq	(-- q) hold parsed input
outputq	(-- q) hold matched phrase or parsed phrases
activity	(-- a) flag: an activity is being defined for a rule
phrase	(terminator -- phrase) create a phrase delimited by terminator
phrases	(initiater \ terminater --) create phrases started by initiator and ended by terminator
stuff-it	(phrase --) stuff a phrase into the inputq
method	(-- method) create a method
in-phrase	(terminator --) create a phrase in a method
add-rule	(sub \ meth \ phr --) add a rule to the current rule-set
add-rules	(sub \ meth --) add a bunch of rules to the rule-set
make-rules	(initiator \ terminator --) create rules
{	(--) define a rule
|	(--) define a rule; alternate syntax
?rules	(--) make sure there is an initial rule words dictionary
>phrase	(strings \ n -- phrase) build a phrase from strings
word-rule	(string \ method --) add a rule for string and method

Rule Matching

checked-rule	(-- a) points to the rule which is being checked
check-word	(-- a) points to a word in the phrase of the rule being checked
rule?	(-- f) is there a rule being matched?
rule-done?	(-- f) is the rule matched?
first-word	(-- s) get first word of the rule
first-sub	(phrase -- s) return first string form substitute phrase
next-word	(--) move to the next of the rule
next-rule	(--) move to the next rule
set-rule	(s --) use string s to select a rule
find-rule?	(-- f) can a rule be found?
get-input	(--) get input in case we are out
restore-input	(--) unparse partially matched input
word-match?	(-- f) does the current word match in the current rule?
primary-word	(--) set primary word for rule checking
following-word	(--) move to following word for checking

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Rule Execution

rule-echo	(-- a) echo rules when executing
do-method	(rule --) execute the method of the rule
substitute	(rule --) substitute the substitute phrase of a rule
run-word	(s --) look up and execute word in string
run-rule	(rule --) execute rule on the stack
rule	(-- a) address of rule which is in the cache
rule-size	(-- a) length of phrase in cached rule
cached?	(-- f) status of rule cache
cache-rule	(--) cache the current rule under inspection
turf-phrase	(--) empty out the phrase queue
run-cache	(--) run the rule which is in the cache
uncache-rule	(--) pull current rule from cache

Rule Navigation

longer?	(-- f) is the current rule longer than the one in cache?
?default	(--) locate and execute the default rule
check-rule	(--) check a rule against the input stream
check-rules	(--) check all rules in a set
check-rule-sets	(--) check all rule sets

File Translator

translate-file	(--) run a file through Timbre
translate	(--) for running Timbre without affecting open files
translate-input	(--) translate input till input exhausted
translator-init	(--) initialize Timbre. This word can be redefined to include other initializations
tld	(“file” | s --) run the following file name through Timbre Use: tld filename
.tld	("file" --) same as tld but print file name
timbre	(-- f) run Timbre on input and return error flag

Introduction
Writing Rules
Writing Rule Sets
Writing FrameWorks
Writing Applications
Debugging FrameWorks

Rules Tools

get-phrase	(-- a) return the address of the match phrase of the current rule
.phrase	(a --) print out the phrase on the stack
.{phrase}	(a --) print out the phrase with curlies around it
.method	(a --) print out the method on the stack
.substitute	(a --) print out the substitute phrase on the stack
.rule	(rule --) print out the rule on the stack
dotrules	(--) print out rules being checked
rule	(--) parse following word and print the rules which match
.q	(q --) print out a queue from the stack
.qs	(--) print out input and output queue
.report	(n \ n \ n \ n --) report results of a SHOWME
showme	(--) show all the effects of executing a rule
dotrule	(rule --) print out a rule
.last-rule	(--) print out the last rule checked
.timbre	(--) print out the status of Timbre
?rmargin	(n -- n) add a new line if n characters won't fit before the right margin
d>table	(dict -- string table \ capacity) translate a dictionary into its string table and its capacity
d>adjunct	(dict -- adjunct table \ capacity) translate a dictionary into its adjunct table and its capacity
.dict	(dictionary --) print out a dictionaries contents
.words	(--) print out the words in the top rule set
.rules	(--) print out all the rules in the top rule set
.set	(a --) print name of rule
.p	(--) print phrase for current rule
file-echo	(-- a) variable to enable file name printing
text-echo	(-- a) variable to enable text echoing

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

9

Index

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

Symbols

! 8-32
!+ 8-32
!- 8-32
" 8-39
8-36
#> 8-36
#>$ 8-47
#arg 8-48
#dos 8-49
#free 8-46
#hash 8-46
#mac 8-49
#s 8-36
#unix 8-49
$! 8-47
$within 8-47
' 8-39
(8-38
* 8-30
+ 8-30
+! 8-32
+$ 8-47
+bits 8-32
+entry 8-46
+in 8-36
+unknown 8-44
, 8-34
- 8-30
-bits 8-32
-branch 8-38
-entry 8-46
. 8-36
." 8-39
.{phrase} 8-53
.b 8-36
.d 8-36
.dict 8-53
.done 8-44
.fields 8-45
.h 8-36
.last-rule 8-53
.line 8-45
.method 8-53
.name 8-45
.p 8-53
.phrase 8-53
.prompt 8-39
.q 8-53
.qs 8-53
.r 8-36
.ram 8-46
.report 8-53
.rule 8-53
.rules 8-53
.s 8-45
.set 8-53
.substitute 8-53
.timbre 8-53
.tld 8-52
.word 8-45
.words 8-53
/ 8-30
/mod 8-30
= 8-31
> 8-31

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

>baron 8-35
>body 8-34
>console 8-49
>file 8-49
>folder 8-49
>i 8-38
>item 8-48
>phrase 8-51
>q 8-32
>r 8-29
>rules 8-50
?adjunct 8-46
?ascii 8-45
?baron-full 8-35
?char 8-38
?cr 8-35
?default 8-52
?dup 8-29
?error 8-37
?execute 8-37
?file 8-42
?find 8-38
?rmargin 8-53
?rules 8-51
?stop 8-45
?wait 8-35
?word 8-38
@ 8-32
@+ 8-32
@- 8-32
{ 8-45, 8-51
| 8-51
} 8-45
}shell 8-47
“ 8-38

Numerics
0 8-30
0= 8-30
0baron 8-35
0branch 8-38
0errors 8-44
0keys 8-36
0q 8-32
0string 8-46
0tib 8-36
2* 8-30
2/ 8-30
2drop 8-29
2dup 8-29

A
abort 8-37
abs 8-31
absolute 8-44
act 8-44
activate-file 8-44
activate-object 8-44
activator 8-44
activity 8-51
add-path 8-43
add-rule 8-51
add-rules 8-51
adjunct 8-46
adjunct-table 8-46
again 8-39
ahead 8-39
align 8-34
aligned 8-34

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

allocate 8-46
allot 8-34
also 8-35
and 8-30
ao 8-44
ao-message 8-44
aobj_file 8-44
append 8-46
array 8-48
asckey 8-46
assume# 8-44
auto-script 8-43

B

baron 8-35
baron> 8-35
base 8-36
begin 8-39
bin 8-36
binary 8-42
bits-differ? 8-31
bl 8-33
blanks 8-33
block 8-43
branch 8-38, 8-44
bump 8-46
bye 8-40
byte-pointer 8-42
bytes 8-43

C

c! 8-32
c!+ 8-32
c!- 8-32
c+! 8-32
c, 8-34
c@ 8-32
c@+ 8-32
c@- 8-32
cache-rule 8-52
cached? 8-52
capacity# 8-46
case 8-45
case-branch 8-38
cd 8-43
cell 8-29
cell-exchange 8-48
cells 8-29
chain 8-47
change-rules 8-50
char 8-38
check-rule 8-51, 8-52
check-rule-sets 8-52
check-rules 8-52
check-word 8-51
cii 8-40
close 8-42
close-all 8-42
cmove 8-33
collect 8-39
collector 8-39
compare 8-31
compile 8-37
compiling 8-37
constant 8-40
container 8-38
copy-rules 8-50
count 8-35
cr 8-35

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

create 8-40

D

d>adjunct 8-53
d>table 8-53
decimal 8-36
default 8-50
default-rule 8-50
definitions 8-35
delete 8-46
depth 8-29
destination 8-49
dict 8-46
dictionary 8-46
dictq 8-50
differ? 8-31
different? 8-46
digit 8-37
digits 8-37
digits? 8-37
dii 8-40
do-method 8-52
does> 8-40
dos 8-49
dos? 8-49
dotrule 8-53
dotrules 8-53
dp 8-34
drop 8-29
dump 8-45
dup 8-29

E

else 8-39
emit 8-35
emit? 8-35
emitq 8-35
empty 8-46
end 8-47
endcase 8-45
endif 8-39
erase 8-33
error 8-37
error-string 8-48
evaluate 8-38
exchange 8-48
exchange-vector 8-48
execao 8-44
execute 8-38
execute-word 8-38
exit 8-39
external-ref 8-45
external-value 8-45

F

file 8-42
file-echo 8-53
file-emit 8-49
file-size 8-42
file-status 8-42
file> 8-49
file? 8-42
filename 8-49
fill 8-33
find 8-46
find-rule? 8-51
find? 8-38
first-sub 8-51
first-word 8-51

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

flush-blocks 8-43
flush-emits 8-35
following-word 8-51
for 8-39
forget 8-40
forth 8-35
free 8-46
free-dictionary 8-46
fullfilename 8-43

G

get-input 8-51
get-phrase 8-53
get-word 8-50

H

hash 8-46
header 8-40
here 8-34
hex 8-36
hold 8-36

I

i> 8-38
if 8-39
image 8-44
immediate 8-40
immediate? 8-44
in 8-36
in-phrase 8-51
in-rules? 8-50
in-what 8-45
init 8-39, 8-43
init-io 8-41
initialize 8-46
inlink 8-45
input 8-36
input-echo 8-43
input-line 8-43
input? 8-38
inputq 8-51
insert 8-46
install 8-40
internal-ref 8-45
internal-value 8-45
interpret 8-38
interpret-object 8-44
interpret-word 8-38

J

junk 8-46

K

key 8-36
key-ask 8-36
key? 8-36
keyq 8-36
kill 8-35

L

l>link 8-34
l>name 8-34
l>tick 8-34
late 8-37
latest 8-34
ld 8-43
less-vector 8-48
less? 8-48
limit 8-34
line-no 8-42
link 8-47
listq 8-35

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

lit 8-38
literal 8-37
load 8-43
load-aof 8-44
load-image 8-44
locate 8-46
locate-file 8-42
longer? 8-52
lookup-symbol 8-44
lowercase 8-37

M

mac 8-49
mac? 8-49
make-rules 8-51
make-word 8-50
max 8-31
memcmp 8-31
method 8-51
min 8-31
mod 8-30
modify 8-42
move 8-33

N

n>link 8-34
n>tick 8-34
name? 8-34
names 8-50
negate 8-30
negative? 8-37
new 8-42
next 8-39
next-rule 8-51
next-rule-set 8-50
next-symbol 8-44
next-word 8-51
nip 8-29
no 8-30
non? 8-30
noname 8-38
not 8-30
number 8-37
number? 8-37
numstr 8-47
nup 8-29

O

oct 8-36
only 8-35
open 8-42
or 8-30
out 8-35
outlink 8-45
outlist 8-45
output 8-49
outputq 8-51
over 8-29

P

p 8-32
pad 8-36
parse 8-37
parser 8-50
parser-rule 8-50
partition 8-48
pathlength 8-43
pause 8-47
peasantq 8-35
peasants 8-45

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

phrase 8-51
phrases 8-51
platform 8-49
pop 8-33
pop-file 8-42
pop-rules 8-50
prefixed? 8-37
previous 8-47
primary-word 8-51
prompt 8-39
prompter 8-39
pull 8-33
push 8-33
push-file 8-42
push-rules 8-50
put-rules 8-50
pwd 8-43

Q

q 8-32
q> 8-32
q? 8-32
queue 8-32
quicksort 8-48
quit 8-39

R

r 8-29
r> 8-29
rdepth 8-29
read 8-42
read-bytes 8-42
read-cell 8-42
read-line 8-42
read-name 8-44
reboot 8-40
recursive 8-40
rehash 8-46
relative 8-44
remove 8-42
rename 8-42
repeat 8-39
repeat-object 8-44
reset 8-40
reset-rules 8-50
resolve-symbol 8-44
restore-input 8-51
restore-io 8-41
rootlength 8-43
rotateq 8-32
rp! 8-29
rp0 8-29
rqsort 8-48
rssq 8-50
rule 8-52, 8-53
rule-done? 8-51
rule-echo 8-52
rule-set 8-50
rule-sets? 8-50
rule-size 8-52
rule-words 8-50
rule? 8-51
ruledict 8-50
ruleq 8-50
rules 8-50
rules-get 8-50
rules> 8-50
run 8-35
run-cache 8-52

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

run-rule 8-52
run-word 8-52

S

same? 8-31
scan 8-37
scan-for 8-37
search-context 8-34
search-dictionary 8-34
search-vocab 8-34
set-base 8-36
set-context 8-34
set-root 8-43
set-rule 8-51
set-timeout 8-47
setup-io 8-41
sh! 8-47
shell 8-47
shell-abort 8-47
shell-end 8-47
shell-out 8-47
shell{ 8-47
shift 8-30
show-peasants 8-45
showme 8-53
sign 8-36
sink 8-49
skip 8-37
skip-prefix 8-37
smudge 8-40
sort 8-48
sort-byadjunct 8-48
sort-dict 8-48
source 8-49
sp! 8-29
sp0 8-29
space 8-35
spaces 8-35
start 8-47
stdin 8-40
stdout 8-40
string 8-39
string-less? 8-48
string-table 8-46
stuff 8-33
stuff-it 8-51
substitute 8-52
swap 8-29
swap-file 8-49
sys 8-41

T

t>link 8-34
t>name 8-34
table-entry 8-45
text-echo 8-53
then 8-39
tib 8-36
timbre 8-52
time 8-47
timeout 8-47
timeout? 8-47
tld 8-52
transferq 8-33
translate 8-52
translate-file 8-52
translate-input 8-52
translator-init 8-52

Symbols
Numerics
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

trim-blank 8-49
tuck 8-29
turf-phrase 8-52
type 8-35

U

u 8-31
u> 8-31
u2/ 8-30
umax 8-31
uncache-rule 8-52
unique 8-50
unix 8-49
unix? 8-49
unknown 8-44
unknowns 8-44
unlink 8-47
unsigned-less 8-48
until 8-39
update 8-43
used? 8-46

V

variable 8-40
version 8-40, 8-50
vii 8-40
vocabulary 8-34

W

w! 8-32
w, 8-34
w.error 8-48
w.errors 8-48
w.messages 8-48
w.notice 8-48
w.notices 8-48
w@ 8-32
watch-stack 8-45
wd 8-43
while 8-39
word 8-38
word-match? 8-51
word-rule 8-51
words 8-45
write 8-42
write-bytes 8-42
write-cell 8-42
write-line 8-42

X

xor 8-30

Y

yes 8-30